Communication par messages

— exemples et langages —

Paradigmes d'interaction entre processus

Primaires :

- producteurs/consommateurs : filtre de tri (merge sort)
- clients/serveurs : moniteur actif
- pair à pair : échange de valeurs

Dérivées :

- coordinateur/travailleurs : multiplication de matrices
- algorithmes synchrones : calcul de routes
- filtres : calcul de nombres premiers
- essai/écho : topologie d'un réseau
- algorithmes à diffusion : horloges logiques
- passage de jeton : exclusion mutuelle distribuée
- serveurs dédoublés : serveurs de fichiers

Échange asynchrone de messages

Clients/serveurs : moniteur actif

Moniteur classique = contrôleur (passif) de l'accès aux ressources

- encapsulation des variables donnant l'état des ressources
- ensemble de procédures pour accéder aux ressources
- exclusion mutuelle entre les appels des procédures
- synchronisation par variables condition

Moniteur actif = processus contrôleur (serveur)

- variables du moniteur = variables locales du serveur
- procédures du moniteur = opérations demandées au serveur
- le serveur traite une seule demande à la fois
- variables de condition simulées explicitement

Exemple 1 : moniteur actif sans variables condition

```
mtype = { op1, op2, ... };
chan request : [N] of { int, mtype, types des argument
chan reply : [M] of { int, type du resultat };

proctype Serveur {
  int clId; mtype opKind; types des arguments args;
  type du resultat res;
  declaration des variables locales du moniteur;
  initialisation des variables du moniteur;
  do :: true -> /* invariant du moniteur */
    request ? clId, opKind, args;
    if :: opKind == op1 ->
            { /* calculs opération; resultat en res */ }
       :: opKind == op2 ->
            { /* calculs opération; resultat en res */ }
       ...
    fi;
    reply ! clId(res)
  od
}
proctype Client(int i) {
  ...
  request ! i, opj, argsj;
  reply ? i(resj);
  ...
}
```

Exemple 2 : allocation de ressources – moniteur passif

```
monitor Allocator {
  int avail = MAXUNITS;
  set units = valeurs initiales;
  cond free;

  procedure acquire(int& id) {
    if (avail == 0) wait(free);
    else avail = avail-1;
    remove(units, id);
  }
  procedure release(int id) {
    insert(units, id);
    if (empty(free)) avail = avail+1;
    else signal(free);
  }
}
```


Exemple 2 : allocation de ressources – moniteur actif

```
mtype = { ACQUIRE, RELEASE };
chan request : [N] of { int, mtype, int };
chan reply : [M] of { int, int };

proctype Allocator () {
  int avail = MAXUNITS;
  set units = valeurs initiales;
  queue pending = initialement vide;
  int clId, unitId; mtype opKind;
  while(true) {
    request ? clId, opKind, unitId;
    if (opKind == ACQUIRE) {
       if (avail == 0) insert(pending, clId);
       else {
         avail=avail-1; remove(units, unitId);
         reply ! clId(unitId); }
    }
    else { /* opKind == RELEASE */
       if (empty(pending)) {
         avail=avail+1; insert(units, unitId);
       } else {
         remove(pending, clId);
         reply ! clId(unitId); }
    }
  }
}
```

Pair à pair : échange de valeurs

Pair à pair :

- un processus = un pair car
- le rôle de chaque processus est identique
- trois topologies de la communication :
  1. centralisée
  2. symétrique ou complètement connectée
  3. en anneau

Exemple : N processus ont chacun une valeur locale ; trouver un algorithme pour que chaque processus trouve les valeurs maximale et minimale des valeurs locales.

Analyser chaque solution selon :

- le nombre de messages échangés
- le délai ajouté par la communication
- la facilité de programmation

Solution centralisée

```
chan values : [N] of { int };
chan results : [N] of { int, int, int };

proctype P0 () { /* coordinateur */
  int v0 = ...; /* valeur locale, initialisee */
  int new, min = v0, max = v0;
  for (int i=1; i < N; i++) {
    values ? new;
    min = (min > new) ? new : min;
    max = (max < new) ? new : max;
  }
  for (int i=1; i < N; i++)
    results ! i(min, max);
}
proctype P (int i) {
  int v = ...; /* valeur locale, initialisee */
  int min, max;
  values ! v;
  results ? i(min, max);
}
```

Solution symétrique

```
chan values[N] : [N] of { int };

proctype P (int i) {
  int v = ...; /* valeur locale, initialisee */
  int new, min = v, max = v;

  for (int j=0; j < N; j++)
    if (j != i)
      values[j] ! v;

  for (int j=1; j < N; i++) {
    values[i] ? new;
    min = (min > new) ? new : min;
    max = (max < new) ? new : max;
  }
}
```


Solution en anneau

```
chan values[N] : [1] of { int, int };

proctype P0 () { /* initiateur */
  int v0 = ...; /* valeur locale, initialisee */
  int min = v0, max = v0;
  values[1] ! min, max;
  /* recupere valeurs finales et transmets a P(1) */
  values[0] ? min, max;
  values[1] ! min, max;
}

proctype P (int i) {
  int v = ...; /* valeur locale, initialisee */
  int min, max;
  /* recupere valeurs temporaires calculees */
  values[i] ? min, max;
  min = (min > v) ? v : min;
  max = (max < v) ? v : max;
  /* envoi le resultat au prochain processus */
  values[(i+1) % N] ! min, max;
  /* recupere le resultat final et retransmet */
  values[i] ? min, max;
  if (i < N-1) value[i+1] ! min, max;
}
```

Échange synchrone de messages

= la réception et **l'envoi** sont bloquants

Avantages :

- mémoire utilisée limitée
- facilité des preuves

Désavantages :

- réduction de la concurrence
- difficulté de la programmation

CSP

- introduit par Hoare en 1978 ;
- formalisme pour spécifier les programmes concurrents avec échange synchrone de messages ;
- utilisé surtout pour étudier les propriétés des programmes.

Occam

- langage de programmation basé sur CSP
- langage machine des *transputers*

Exemple de filtres : calcul de nombres premiers

```
chan sieve[N] : [0] of { int };

proctype S0 () { /* initiateur */
  int p = 2;
  for (int i=3; i < L; i++)
     sieve[0] ! i;
}
proctype S (int i) {
  int p; /* premier local */
  int next;
  sieve[i-1] ? p;
  do :: sieve[i-1] ? next ->
        if :: next % p != 0 -> sieve[i+1] ! next
           :: else -> skip
        fi
  od
}
proctype End () {
  int n;
  do :: sieve[N-1] ? n -> ...
  od
}
```

Linda

- approche originale de la programmation concurrente
- pas un langage mais une collection de primitives
- éléments de base :
  - espace de tuples (TS) = mémoire associative typée
    `("prime",  4, 7)`
  - 3 primitives d'accès :
    * IN : prise (bloquante) d'un tuple
    * OUT : dépôt d'un tuple
    * RD : lecture (bloquante) d'un tuple
  - 2 prédicats d'accès non-bloquants :
    * INP : prise d'un tuple s'il existe
    * RDP : lecture d'un tuple d'il existe
  - 1 primitive pour la création de processus :
    * EVAL : création d'un processus pour chaque expression à évaluer

Exemple : calcul de nombres premiers

Organisation :

- coordinateur/travailleurs
- taches données par le TS

Typage du TS :

```
("stop")                /* arret des travailleurs */
("candidate", int)      /* tache a faire */
("result", int, int)    /* resultat obtenu */
("prime", int, int)     /* nombre premier */
```

Demandez-moi de vous le montrer.

Java

Package java.net = primitives de communication par messages

- communication non fiable (UDP) par paquets
- communication fiable (TCP) par sockets
- communication niveau application (URL)

Communication fiable (TCP) par *sockets* :

- classe ServerSocket pour l'attente de connexion
- classe Socket pour la connexion
- communication par flou de messages :
  - communication permanente (jusqu'à la fermeture)
  - flou de données bidirectionnel
  - flou fiable et ordonné

Exemple : serveur et client d'écho